# Understanding PARSEC Performance on Contemporary CMPs

Major Bhadauria, Vincent M. Weaver
*Cornell University*
{*major* | *vince*}*@csl.cornell.edu*

Sally A. McKee
*Chalmers University of Technology*
*mckee@chalmers.se*

## Abstract

*PARSEC is a reference application suite used in industry and academia to assess new Chip Multiprocessor (CMP) designs. No investigation to date has profiled PARSEC on real hardware to better understand scaling properties and bottlenecks. This understanding is crucial in guiding future CMP designs for these kinds of emerging workloads. We use hardware performance counters, taking a systems-level approach and varying common architectural parameters: number of out-of-order cores, memory hierarchy configurations, number of multiple simultaneous threads, number of memory channels, and processor frequencies. We find these programs to be largely compute-bound, and thus limited by number of cores, micro-architectural resources, and cache-to-cache transfers, rather than by off-chip memory or system bus bandwidth. Half the suite fails to scale linearly with increasing number of threads, and some applications saturate performance at few threads on all platforms tested. Exploiting thread level parallelism delivers greater payoffs than exploiting instruction level parallelism. To reduce power and improve performance, we recommend increasing the number of arithmetic units per core, increasing support for TLP, and reducing support for ILP.*

## 1 Introduction

The PARSEC (*Princeton Application Repository for Shared-Memory Computers*) benchmark suite [4] attempts to represent both current and emerging workloads for multiprocessing hardware. The suite is intended to help drive CMP design research in both industry and academia. For these applications to help guide future designs, users must first understand which hardware resources are critical for desired system behavior, particularly as applications need to scale with increasing number of processing cores. Studying application behavior on existing systems representative of contemporary design trends is a good starting point for developing the foundations of this understanding. We therefore evaluate PARSEC on several contemporary CMP platforms, each representative of different design choices that can greatly affect workloads. PARSEC applications exhibit different memory behaviors, different data sharing patterns, and different workload partitions from most other benchmark suites in common use (see Section 2), which is precisely why we choose to study them on current hardware in order to learn how to better design future hardware for user and commercial workloads, as opposed to just the commonly studied HPC workloads.

We investigate several aspects of contemporary CMPs: memory speeds, SMT performance, out-of-order execution, and performance scaling with increasing number of threads. We specifically examine current micro-architectural resource bottlenecks. In doing so, we identify:

- bottlenecks in current systems for particular PARSEC applications;
- machine aspects that have little effect on PARSEC application performance (and thus are not good candidates for future architectural enhancement for these types of workloads); and
- resources crucial to good scaling properties for PARSEC applications.

The PARSEC applications exhibit differing memory behaviors: some are bandwidth limited, performing clustered accesses to large amounts of data (with little locality); others are latency limited, with some enjoying good locality (which the memory subsystem can exploit), and others exhibiting scattered accesses with little locality (impeding any memory backend from masking latencies).

Making good use of memory resources is necessary, but more important is these applications' ability to exploit parallelism. In particular, we find that exploiting thread-level parallelism (TLP) over instruction-level parallelism (ILP) greatly benefits these workloads. Larger caches have little effect on performance, and thus we conclude that design time and transistors are likely better spent on increasing logic to improve TLP, reducing speculative logic intended to exploit ILP, and prefetching into local caches (since cache accesses tend to have high locality, and misses do not cause invalidations in other caches).

## 2 Related Work

Most architectural studies, including workload studies, use simulation to gather data about the applications in question. If done well, this can provide information that cannot be obtained by running applications on native hardware. On the other hand, running codes on real platforms provides some information unobtainable (or unreliable) from simulation. Whichever method is used, care must be taken to ensure accuracy of results (e.g., OS noise must not distort hardware measurements, and simulation results — especially from partial simulations — are subject to both inherent simulator error as well as methodological errors). Here we focus on the applications and kernels used to study existing and proposed hardware performance. A discussion of using simulation versus hardware performance monitoring counters is beyond the scope of this paper [18].

Bienia et al. [4] study PARSEC in detail using functional simulation, and (separately) compare PARSEC to the SPLASH-2 benchmarks [3, 19], finding that PARSEC applications exhibit more intense sharing among threads, exploit a pipelined programming model for several programs, and use larger memory footprints that cause higher cache miss rates over other benchmark suites. We gather additional results, using real hardware to cover metrics that cannot be obtained via purely functional simulation.

The NAS Parallel Benchmarks [2] (NPBs) represent early efforts to evaluate performance of highly parallel platforms. The NPBs address shortcomings of previous benchmarks, including the Livermore Fortran Kernels [13] (the "Livermore Loops"), LINPACK (Fortran subroutines to solve a variety of linear equations and least-squares problems), and LINPACK's successor LAPACK (the "Linear Algebra Pack", which explicitly exploits shared memory). These suites largely target vector supercomputers, and are insufficient (especially with respect to problem size) for studying scalability of emerging supercomputers. Furthermore, full application benchmarks are difficult to port and lack automatic parallelization tools (or they did at the time of benchmark introduction). The NPBs thus emerged as the benchmarks of choice for evaluating large scale parallel machines. They also enjoy widespread use in the evaluation of proposed machine designs (including CMPs), but specifically target scientific computations (e.g., large-scale computational fluid dynamics applications). Like SPLASH-2 [19], they exhibit different performance behaviors from PARSEC, which explicitly tries to evaluate machines not solely designed for scientific supercomputing.

MiBench [9], MediaBench [11] (with an advertised upgrade coming soon), and CSiBE [6] are all publicly available suites suitable for embedded systems research, including CMP studies, but they target another niche market. In contrast, PARSEC targets general-purpose computing.

Alam et al. [1] characterize performance of scientific workloads on multicore chips. Iyer et al. [10] explore resource bottlenecks for commercial Java servers, examining tradeoffs of cores, caches, and memory.

All non-PARSEC studies discussed herein investigate special-purpose workloads; we investigate a broader range of workloads (always on actual hardware, and never in simulation) to determine which architectural features are most crucial for *general-purpose* CMPs.

## 3 Experimental Setup

We evaluate the PARSEC codes according to a set of metrics spanning CMP performance, and, in particular, affecting scaling properties as applications grow to greater numbers of threads. We consider three properties with respect to cache performance: number of evictions compared to total accesses, bus invalidation requests (but not snoop traffic), and data bus traffic from writebacks and core-to-core transfers. Since memory tends to bottleneck applications, we examine two metrics with respect to DRAMs and thread performance: reducing bandwidth and reducing memory speeds. With respect to throughput, we examine chip traffic ratios (main memory accesses per retired instruction) to quantify how dependent an application is on memory. Most importantly, we measure performance scaling as number of threads increases for each application. We examine bottlenecks, and identify whether they are software or hardware artifacts. We investigate performance of scaling number of threads via Simultaneous Multithreading (SMT) versus increasing individual cores. With respect to how micro-architecture resources affect performance, we examine processor stalls due to lack of resources, including Re-Order Buffer (ROB) hazards, unavailable Reservation Station (RS) slots, unavailable Load/Store Queue (LSQ) slots, and contention for Floating Point (FP) units.

PARSEC consists of 12 applications from Princeton, Intel, and Stanford. V1.0 of the suite represents a diverse set of commercial and emerging workloads. Table 1 lists the applications; for more information, please see Bienia et al. [4]. We use publicly available tools and hardware to ensure reproducible results. We use `gcc 4.2` C and C++ compilers with Linux kernel 2.6.22.18. The only exception is our Solaris machine, which uses the Sun C compiler. We compile in 64-bit mode using statically linked libraries. All experiments use full native input sets.

We test a range of system configurations that vary in memory hierarchy, system frequency, and micro-architecture. Table 2 lists machines and relevant architectural parameters. *Mem Ch* indicates number of memory channels. All processors have inclusive L1 and L2 caches. None of the processors has an L3 cache except the *Phenom4*, which contains an exclusive 2MB shared L3 cache.

| Name | Description |
|---|---|
| blackscholes | calculates portfolio price using Black-Scholes PDE |
| bodytrack | computer vision, tracks 3D pose of human body |
| canneal | synthetic chip design, routing |
| dedup | pipelined compression kernel |
| facesim | physics simulation, models a human face |
| ferret | pipelined audio, image and video searches |
| fluidanimate | physics simulation, animation of fluids |
| freqmine | data mining application |
| streamcluster | kernel to solve the online clustering problem |
| swaptions | computes portfolio prices using Monte-Carlo simulation |
| vips | image processing, image transformations |
| x264 | H.264 video encoder |

**Table 1.** PARSEC Overview

| Platform | Arch | Freq GHz | Cores | L1 D$ KB | L2 KB | FSB MHz | Mem Ch |
|---|---|---|---|---|---|---|---|
| Athlon1 | x86 | 1.8 | 1 | 64 | 256 | 400 | 1 |
| Atom1 | x86 | 1.6 | 1 (2 SMT) | 24 | 512 | 533 | 1 |
| Conroe2 | x86_64 | 1.8 | 2 | 64 | 2048 | 800 | 2 |
| NiagaraII-8 | SPARC | 1.15 | 8 (64 SMT) | 8 | 4096 | 800 | 4 |
| P4Classic | x86 | 2.4 | 1 | 16 | 512 | 533 | 1 |
| P4DSMT | x86_64 | 3.46 | 4 (8 SMT) | 16 | 2048 | 1066 | 2 |
| Phenom4 | x86_64 | 2.2 | 4 | 64 | 512 | 1800 | 2 |
| Xeon8 | x86_64 | 2.32 | 8 | 64 | 4096 | 1066 | 4 |

**Table 2.** Machine Parameters

*Athlon1* is a traditional single-core AMD processor. *Atom1* is an Intel in-order dual-issue 32-bit processor that supports SMT. *P4Classic* is a traditional single-core Intel Pentium 4 processor. *P4DSMT* is a Pentium 4 dual-core CMP with private L1 and L2 caches. *Phenom4* is a quad-core CMP with private L1 and L2 caches and a shared 2MB L3 cache. *Conroe2* is a dual-core CMP with private L1 and shared L2 caches. *Conroe4* (not listed) is two Conroe2 CMPs connected as a SMP on a single-chip. *Xeon8* has the same architecture as *Conroe4*, but uses FB-DIMM memory instead of DDR. *NiagaraII-8* is an eight-core CMP with private L1 and shared L2 caches, and FB-DIMM memory.

We gather data via hardware performance counters (both per processor and system wide). We use the `pfmon` tool from the perfmon2 performance counter package [7] to read counter values, and the Linux `time` command to gather actual elapsed execution time. We find little variation between multiple runs for most benchmarks, and treat differences of a few percent as OS noise when comparing architectural choices. We run programs locally since some exhibit significant variation (`bodytrack`, `dedup`, `facesim`, `vips`, and `x264`) when run over the network.

# 4 Summary of Findings

We investigate architectural performance across many real hardware configurations. The main difference that we find compared to the simulation studies of Bienia et al. [4] is that the applications are not limited by memory, as their simulator indicates, but instead by core-to-core communication costs. Main memory speeds do not have a large influence on our measured performances, and neither does out-of-order execution or other support for instruction-level parallelism. The experiments presented below have two purposes: to help users understand PARSEC performance on contemporary CMP platforms, and to help them understand the differences in what simulation experiments reveal versus what hardware performance studies can show. Obviously, when creatinging completely new designs, simulation is required, but knowing application characteristics on existing hardware provides guidance on how to best design such new platforms.

For `fluidanimate`, we omit non-power-of-two thread configurations: they are unsupported. Likewise, for `facesim`, we omit five and seven thread configurations. `Facesim` and `ferret` do not run on SPARC, and are thus omitted from our *NiagaraII-8* evaluation.

## 4.1 Cache Performance

We use the *Phenom4* platform to examine the waterfall effect of data accesses on a three-level cache hierarchy, consisting of inclusive private L1 and L2 caches, and an exclusive shared L3 cache. This analysis is important in determining the effectiveness of a multi-level cache hierarchy. Figure 1 graphs miss rates, where each cache's misses are normalized to total data cache accesses for that level of cache (not including snoops, instruction cache accesses, hardware prefetches, or TLB lookups). Misses per thousand retired instructions (MPKI) are also graphed on the secondary y-axis. L1 and L2 MPKI are normalized to instructions retired for a single core, and L3 MPKI are normalized to total instructions for the entire CMP, since the L3 cache is shared among cores. All benchmarks have low L1 miss rates, the highest being 5% for `canneal` and 2% for `bodytrack`. These miss rates have a trickle-down effect for `canneal`, causing high L2 and L3 misses. Most of these are cold, capacity, and coherence misses, since Bienia et al. [4] find the benchmarks to have large working sets, and our analysis of increasing L2 cache sizes finds minimal reductions in miss rates. `canneal` and `streamcluster` have almost the same number of misses for L1 and L2 caches, indicating the working set does not fit in the *Phenom4's* 64KB L1 or 512KB L2. The L3 cache does not benefit these benchmarks: it exhibits very high miss rates for half the benchmarks. Temporal locality does not extend to the L3, thus for the L3 to capture L2 misses it must be several orders of magnitude larger [4]. We verify the high temporal locality of the L2 accesses by examining numbers of L2 line evictions compared to L2 accesses.

Figure 2 graphs cache data evictions of valid data normalized to total cache accesses for four threads. Evictions

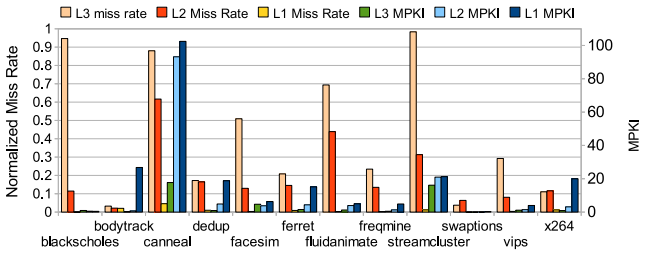


**Figure 1.** Phenom4 4-Thread L1,L2,L3 Cache Performances

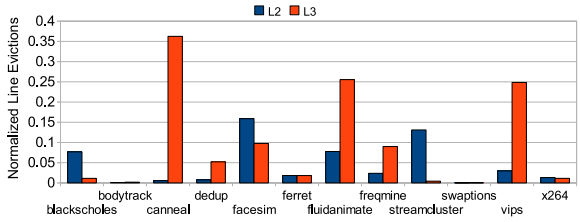


**Figure 2.** Phenom4 4-Thread L2, L3 Data Evictions

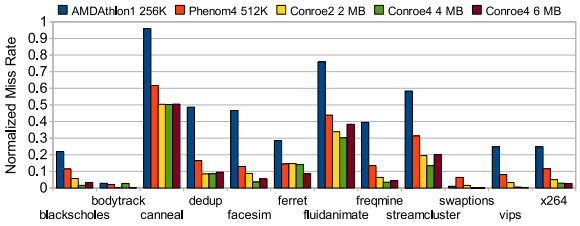


**Figure 3.** L2 Cache Miss Rate Scaling for 1 Thread (256KB-6MB Caches)

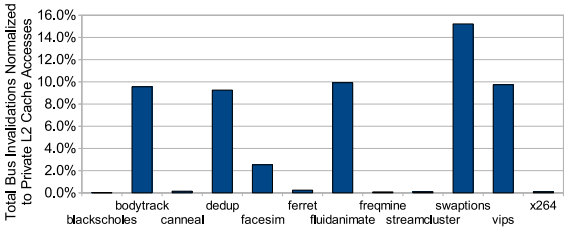


**Figure 4.** Bus Invalidations Normalized to L2 Accesses For 1 Thread (Xeon8)

of valid data are lower than cache misses: the L2 average miss rate is 20%, but the eviction rate is only 5%. Data brought into cache are not evicting valid data, but cache lines invalidated from coherence checks. If requested data are originally evicted due to invalidation from another processor, increasing cache sizes will not help, although reducing line sizes might, if false sharing is the problem. L2 cache accesses exhibit high temporal locality, since application working sets do not fit in the L2 [4], and eviction rates are very low (less than 5% of all L2 cache accesses).

Figure 3 graphs cache miss rates for L2 caches on various machines, ranging from 256KB to 6MB, running a single thread. *Conroe4* 6MB is similar to the *Conroe4* 4MB machine, but it has a 24-way associative cache instead of the 16-way associative cache used for the *Conroe4* 4MB machine. `Canneal`, `dedup`, `ferret`, and `fluidanimate` suffer approximately the same number of misses. Their working sets are sufficiently large that the caches miss regardless of size. On average, doubling L2 cache size from 2MB to 4MB reduces the miss rate from 13% to 11%. Looking at tradeoffs among cache sizes and numbers of processors, 512-1024KB seems the best size-for-area tradeoff. The extra area and power saved can be better used by increasing number of cores (see Section 4.5).

We examine bus invalidations sent by each core to the shared bus. We omit snoops, since they can be determined from L2 miss rate and number of cores. Bus invalidation requests are sent when a cache needs to modify a shared cache line or needs to write a line it lacks. Bus invalidations are important performance factors in designs with larger caches (assuming a shared bus organization among local cores). Figure 4 graphs total bus invalidations for *Xeon8* running eight threads. Bus invalidations are normalized to cache accesses for only one of the shared L2 caches. We choose to use the performance of one of the shared L2 caches — rather than all of them — for normalization, since every time there is an invalidation request from any of the cores, each of the L2 caches needs to check its tags, creating local traffic on every L2 cache connected to the shared bus. Clearly, future CMPs must manage cache invalidations since they contribute to a large portion of cache tag accesses, they will grow with increasing number of cores, and they may not be mitigated by increasing cache sizes. Cache invalidations play a role in cache misses, as well as cache power consumption. However, since snoops only access L2 tags, power overhead may be negligible, depending on system architecture and process technology [14].

### 4.2 Benchmark Sensitivity to DRAM Speeds and Bandwidth

We vary memory speeds and number of memory channels, and examine effects on performances. Figure 5 graphs single-thread benchmark performances on *Phenom4* normalized to a 400 MHz FSB, dual-channel memory configuration (the stock configuration). We examine single thread performances, since memory sensitive benchmarks might bottleneck at higher thread counts regardless of memory speed, masking contention. We explore reducing FSB from 400 MHz (800) to 266 MHz FSB (533), as well as going

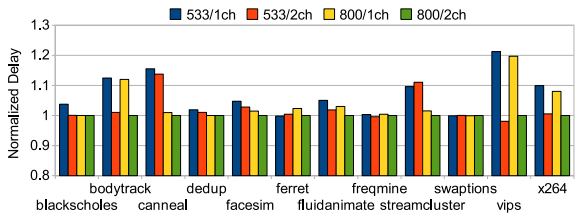


**Figure 5.** Phenom4 1-Thread Performance with Varying Memory Parameters (Lower Is Better)

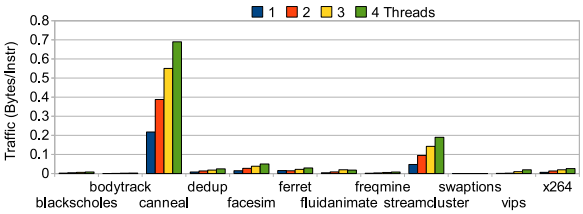


**Figure 6.** Phenom4 Thread Traffic (Bytes/Inst) at Varying Thread Counts

from dual-channel (2ch) to single-channel (1ch). These are speed and memory channel reductions of 33% and 50% respectively. Reducing FSB speed and number of memory channels reduces maximum bandwidth by 67%, since reducing FSB speeds degrades latency and bandwidth. Results show benchmarks are dependent on different dimensions of memory for performance. Figure 5 shows that `bodytrack`, `vips` and `x264` are sensitive to reductions in number of memory channels. Halving the number of memory channels increases delays by 12%, 21%, and 10%, respectively. `Canneal` and `streamcluster` are sensitive to memory latency, suffering delay increases of 16% and 10%, respectively. Increased memory speeds and bandwidths add overhead to system cost by requiring higher speeds for the motherboard's FSB and memory modules, as well as requiring twice the number of traces on the motherboard for dual-channel memory.

Figure 6 graphs total memory traffic in bytes per instruction committed. It shows there is ample bandwidth available for the three memory channel sensitive applications (`bodytrack, vips, x264`) to prefetch data (beyond what is currently performed by the hardware) and mask the memory gap on systems with few memory channels. `canneal` and `streamcluster` require low-latency off-chip accesses, since (as we saw in Figure 5) they are more sensitive to latency than bandwidth. Other benchmarks have sufficiently low traffic demands at low thread counts that memory performance is not an issue.

The PARSEC suite is bound more by computation than memory, since our reductions in memory speeds and bandwidth do not lead to linear degradations in performance. We verify this by examining performance when frequency is doubled from 1.1 GHz to 2.2 GHz on the *Phenom4* platform with four threads. We omit the performance graph due to space constraints, however we find increasing frequency leads to similar increases in performance for half the benchmarks. The exceptions are the benchmarks listed in Table 3. Scaling memory speeds and bandwidth has little effect on performance, but processor frequency has a direct and linear effect. Whether this changes at higher thread counts remains to be seen.

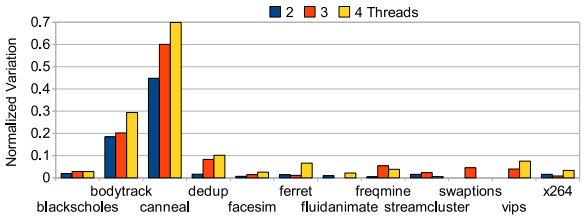


**Figure 7.** Phenom4 Instruction Variation from 1-4 Threads

### 4.3 Thread Scaling Overheads

We investigate two metrics that affect thread performance scaling: the instruction overhead from increasing thread counts and the instruction variation among threads. We do not examine the overhead of dynamic thread synchronization (i.e., contention for critical sections), but indicate in Table 3 when it plays a role in scalability. We analyze instructions retired from one to four thread configurations, and find the count does not change for any benchmark except `canneal`. This shows that the majority of the PARSEC suite has little increased instruction overhead from parallelization. While not graphed due to space constraints, `canneal` exhibits variation of 10-20% in total retired instructions due to its non-deterministic method of using multiple threads to find the most efficient solution, which leads to data races changing program behavior.

Serial portions of a program change the number of retired instructions among cores, assuming the thread does not migrate between processors. Figure 7 graphs variation between threads, normalizing results for the processor that retires the fewest instructions to those of the processor that retires the most. The higher the variation, the higher the percentage of total instructions run by a single processor. There is some inherent error with this technique, since some serial portions can be running on one core and then migrate to another. We compare error across multiple executions, and find variation to be negligible. `bodytrack` and `canneal` both have large variations in retired instructions, and these

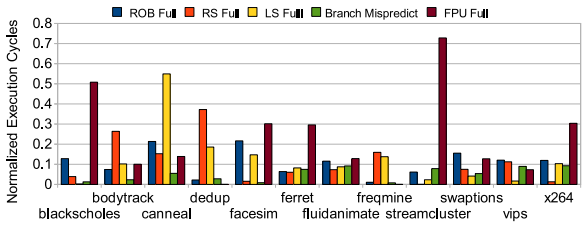


**Figure 8.** Resource Stalls for Phenom4 with 4 Threads

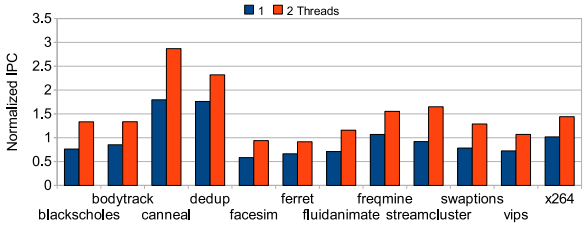


**Figure 9.** Atom1 IPC Performance Normalized to P4Classic (Higher Is Better)

grow with number of threads. As CMPs increase in number of cores, serial portions of these benchmarks will bottleneck performance. One solution is a heterogenous architecture, where one processor executes at a higher frequency [17] or has a wider pipeline.

### 4.4 Micro-architecture Design Choices

We investigate causes of processor stalls, improvements from out-of-order execution and improvements from leveraging SMT. The performance of some of these may change with system design choices (such as the LSQ filling up from longer memory latencies), core metrics such as branch mispredictions and FPU stalls should remain unchanged. Figure 8 graphs clock cycle stalls due to a lack of entries in the reorder buffer (ROB Full), reservation stations (RS Full), and load/store buffers (LS Full), as well as branch mispredictions, and FPU backlogs. `blackscholes` and `streamcluster` spend over 50% of their cycles waiting for the FPU. This is attributed to 43% and 52% of all operations in `blackscholes` and `streamcluster` being floating point [4]. FPU stalls account for 30% of stalls for more than half the floating point benchmarks (three PARSEC benchmarks have few or no floating point operations). This is surprising, since bandwidth was previously thought to be the biggest performance limitation [4], but our memory analysis at low thread counts finds otherwise.

We investigate performance of OoO cores, but omit examining the quantitative tradeoffs of added die area and power consumption, since they are process-technology specific. OoO scheduling can lead to power inefficiencies due to increased number of logic units and execution of misspeculated instructions and load replays. For perspective, the AMD K8 processor with 128 KB of L1 cache and 512 KB of L2 has 68.5 million transistors. The Pentium 4 with 20 KB of L1 (12 KB trace cache and 8 KB Dcache) and 512 KB of L2 cache has 55 M transistors [5]. The recently released in-order Intel Atom processor with 32 KB of L1 and 512 KB of L2 has 47 M transistors [8]. While the reduced transistor count is not significant between in-order and OoO, power consumption is considerably lower (as has been the case with other in-order designs from ARM and Sun [15]). The Sun Niagara 2 with eight cores, eight SMT threads per core, and 4 MB of shared L2 across all cores has 500 M transistors (with).

In Figure 9, IPC of *Atom1* is graphed, normalized to the OoO *P4Classic*. We choose the *P4Classic* processor for comparison with the *Atom1* because both designs have the same amount of L2 cache, but the OoO processor has a complex core that performs a significant number of speculative operations to enhance performance. Although both are Intel designs, they have different pipeline depths and are targeted for different applications (desktop vs. mobile). The figure shows IPC for both one and two threads on the SMT processor. Instances where benchmarks are bound by memory or little ILP can be extracted from the code, the in-order processor is able to achieve as good as or better IPC than the OoO one. Even compute intensive benchmarks such as `blackscholes` only exhibit a 25% degradation in IPC on an in-order core. Enabling the second thread for the *Atom1* processor results in a 55% average improvement in IPC, and a higher IPC than the OoO processor for most benchmarks (the only exceptions being `facesim` and `ferret`). These results indicate that higher performance might be possible by replacing a single OoO core by several in-order cores or by implementing SMT support within a single core. Multithreaded programs allow us to trade ILP for explicit thread level parallelism, reducing the burden on the hardware to extract parallelism at run-time. This can lead to improved power efficiencies assuming programs scale as well with threads as they do with OoO cores. In our results, we find this to be the case, with SMT scaling of just a single extra thread increasing performance over using an OoO core.

We test SMT performance on *P4DSMT* (8 SMT) and *NiagaraII-8*(64 SMT), and comparing against the micro-architecture stalls on the *Phenom4*. Although specific resource sizes differ between the AMD, Intel, and Sun CMPs, the benchmarks still stress the same areas across platforms. Threads can stall due to memory latencies or thread synchronization. SMT can leverage this to achieve performance improvements by executing some threads while others are stalled. Figure 10 shows speedups of bench-

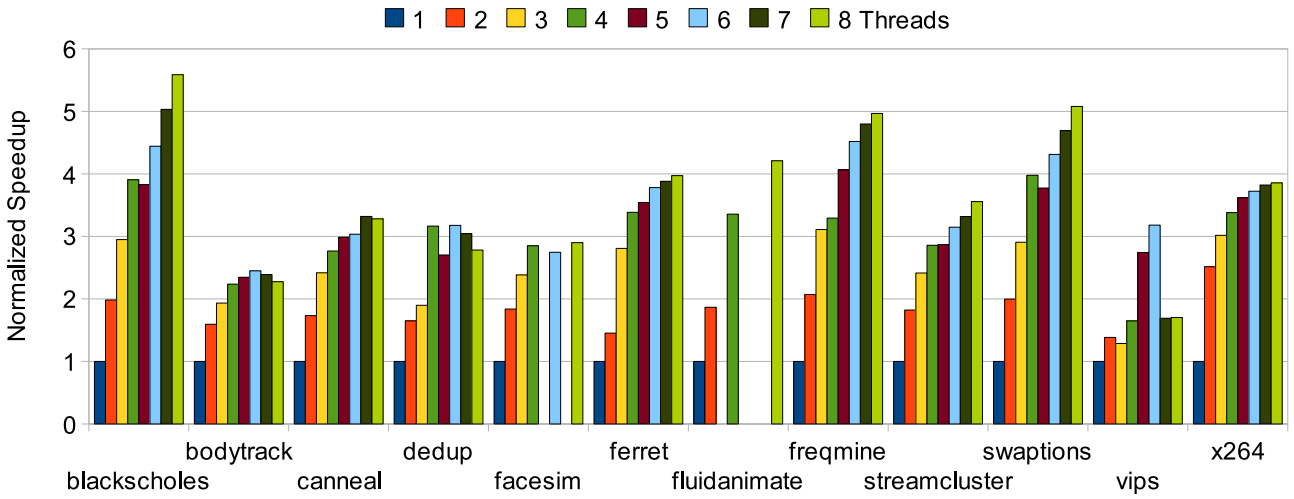


**Figure 10.** P4DSMT 8-SMT Machine at 1-8 Threads (5-8 Virtual)

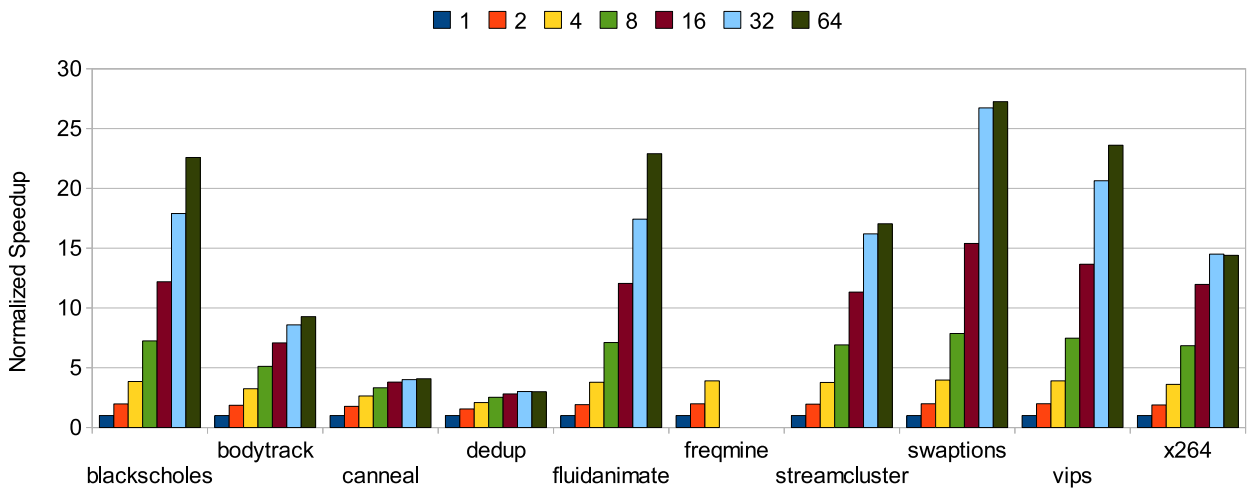


**Figure 11.** NiagaraII-8 Performance Scaling 1-64 Threads (16-64 Virtual) `facesim`, `ferret` Omitted Due to Incompatibility

marks as threads increase from one to eight on four physical processors. We ignore non-SMT performance for the moment, and address it in the next section. Benchmarks `blackscholes`, `bodytrack`, `fluidanimate`, `freqmine`, `swaptions`, and `x264` achieve significant performance gains with virtual cores. SMT usage gives `freqmine` a performance boost since other threads can be swapped into the core when one thread is stalled waiting data. `swaptions` does well with SMT due to excess architectural resources available for extra threads.

On the Intel architecture employed by *P4DSMT*, architectural resources are either duplicated, partitioned, or shared. The caches are shared among threads, and instruction pointer and thread state information is duplicated. However the ROB, LS buffers, and RS are dynamically partitioned between threads, so the SMT processor has half of these resources per thread compared to running a single thread [16]. This bottleneck leads to reduced performance for benchmarks `canneal`, `dedup` and `facesim` (bottlenecks are shown in Figure 8).

`blackscholes` is an FP-limited benchmark [4], as is `streamcluster` for large input dimensions. Due to the low ILP in these benchmarks, some FPUs are idle, which means they can then be utilized by hyper-threading. In contrast, `canneal` performs more data swaps than actual arithmetic. While swaps are evaluated, locks are not held, changing execution paths if one of the swap elements has been replaced by another thread [4]. Unfortunately the LS buffer size is reduced when running multiple threads on the SMT, leading to `canneal`'s sub-par performance gains. `Dedup` is primarily an integer application, using the integer units for hash computations, checksums, and compression. When the integer units are full, instructions queue up in the reservation stations. Once a sufficient backlog exists, the processor must stall to wait for the reservation stations to empty. Increasing threads does not help since the number of reservation station entries and integer units fail to increase in tandem. `x264`'s parallelization scheme contains dependencies between the current frame and the previous frames that are split across threads. When one thread is stalled due to the FPU, it ends up slowing down other threads, as well. Bienia et al. [4] recommend that more threads than cores be used to speed up this benchmark, and we find that this does help for multiple virtual cores (SMT).

Figure 11 shows results for the *NiagaraII-8* platform scaling from 1-64 threads. Leveraging SMT on the SUN platform results in performance trends similar to the *P4DSMT* platform. Since each of the *NiagaraII-8* physical cores only have two ALUs and one FPU shared among threads, there is contention for resources at high thread counts, resulting in smaller gains for some benchmarks. Gains are lower on the *P4DSMT* platform than the *NiagaraII-8*, since the OoO platform is already keeping execution units busy from other independent instructions, therefore there are fewer resources available for executing instructions from another thread. Thread synchronization overhead is not the cause of this poor thread scaling behavior, since gains are higher on other platforms. The *NiagaraII-8* uses round-robin scheduling of the instruction issue queue to achieve fine grain multi-threading, but not true SMT since instructions across different threads are not executed simultaneously. Future SMT designs can provide greater benefits if they include more resources per thread, so when the number of threads increase, there are adequate execution units and data queues available. Based on improvements from SMT, and the code structure, OoO execution fails to leverage existing parallelism to the level that SMT greater than two threads achieves. To remain within a given power envelope, the number of cores should be reduced, while increasing SMT resources per processor.

### 4.5 Thread Scaling Performance

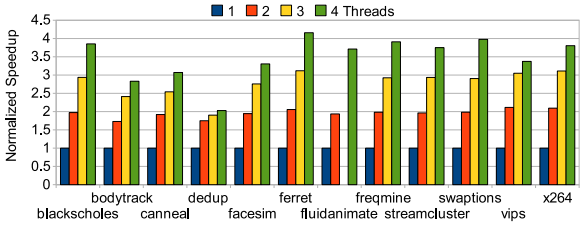


**Figure 12.** Phenom4 Performance Scaling 1-4 Threads

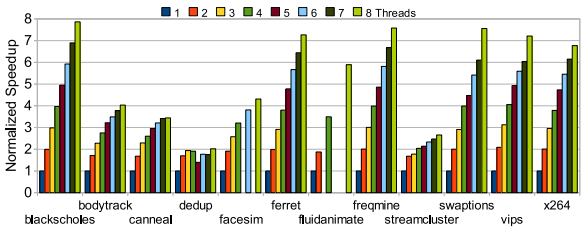


**Figure 13.** Xeon8 Performance Scaling 1-8 Threads

It would be difficult to gain insight from comparing raw execution times across platforms, since that have different frequencies, different cache sizes, different microarchitectures, and even different ISAs. However, by normalizing the results for each platform to single-thread performance, we can observe scaling properties across platforms. Figures 10, 11, 12, and 13 graph performance for *P4DSMT*, *NiagaraII-8*, *Phenom4*, and *Xeon8* normalized to their single thread performances. Benchmarks `blackscholes`, `ferret`, `freqmine`, `swaptions`, `vips` and `x264` scale well across all platforms. This is notable since *P4DSMT* and *Xeon8* require communicating

over the FSB for all core-to-core communication, and lack on-chip memory controllers. Depending on the application, it might be cost effective to use SMPs, since very large CMPs lead to smaller chip yields. `streamcluster` scales better on large CMP designs such as *Phenom4* and *NiagaraII-8* than on *P4DSMT* or *Xeon8*. This is due to `streamcluster`'s heavy data traffic (the most core-to-bus data transfers, and the second most L2 snoops). `streamcluster` is also sensitive to memory latency, as illustrated earlier. The SMT improvements on *NiagaraII-8* are due to the swapping of threads when one thread is waiting on communication. This saturates at higher thread counts when all threads are stalled waiting for the single FPU on the core. `facesim` scales badly at high thread counts, surprising since it has little data exchange [4] and is not bound by main memory.

We use the `oprofile` and HPC Toolkit profiling tools to determine execution cycles each thread spends within different functions. We compare profiles at different thread counts to determine which functions scale and which ones do not, and examine source code to locate bottlenecks [12]. We only examine the programs exhibiting the worst scaling: `bodytrack`, `canneal`, `dedup`, `facesim`, `fluidanimate` and `streamcluster`. Table 3 lists worst scaling programs and sources of the poor scaling. Biggest sources are unequal workload distribution (serial portions), thread synchronization, and bus traffic.

One method for accommodating workload imbalances is to use heterogenous high performance cores. The high performance core can be used when program performance is dependent on a single thread, and weaker, power-efficient cores can be used for parallel portions. Cache prefetching and speculative execution can reduce the performance penalty from thread synchronization and locks. Bus traffic can be alleviated by wider, independent, or faster communication buses. `fluidanimate` scales well on the *NiagaraII-8*, but fails to scale well on *Xeon8* after four threads. This is due to the shared cache on the *NiagaraII-8* having lower core-to-core latencies, a smaller gap between memory and CPU, and the *NiagaraII-8*'s spending more time processing non-critical sections (less thread synchronization overhead).

While current hardware plays a role in limiting the scalability of some applications, half of the PARSEC suite is sufficiently computation bound to scale well regardless of platform. Unless the aforementioned scaling bottlenecks are corrected, several of these applications will fail to scale on future hardware. Based on our evaluation, off-chip memory traffic will not bottleneck systems as much as cross-thread communication, requiring better methods of sharing data across cores. Current native four (*Phenom4*) and eight core (*NiagaraII-8*) designs are insufficient in this regard.

## 5 Conclusions

Half of the PARSEC applications enjoy performances that scale approximately linearly for up to eight threads. Most are computation bound, although some require significant memory bandwidth or are sensitive to memory latencies. Nonetheless, the area and power required for larger caches are impractical and can be better spent on other system resources. For instance, limits to ILP (from insufficient hardware resources or properties inherent in the software) and TLP, micro-architectural hazards, and core-to-core communication costs have a greater impact on performance. Exploiting TLP (through adding cores or SMT support) at the thread counts we study yields more benefit than trying to exploit ILP through speculation, even though some applications unavoidably fail to benefit from increasing TLP due to critical section locks and large serial portions of code.

## 6 Acknowledgments

We would like to thank Gary Tyson and Stephen Hines at Florida State University for the use of their Sun Niagara 2 hardware. Without it, our insights on the Sun SPARC platform would not have been possible. We also thank Chris Fensch from the University of Cambridge for his patches to enable execution on the SPARC platform. This material is based upon work partly supported by the National Science Foundation under Award Numbers 0444413 and 0444413, as well as by a Natural Sciences and Engineering Research Council PGS-D Fellowship.

## References

[1] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *Proc. IEEE International Symposium on Workload Characterization*, pages 225–236, Oct. 2006.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[3] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proc. IEEE International Symposium on Workload Characterization*, pages 47–56, Sept. 2008.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Oct. 2008.

[5] D. Deleganes, J. Douglas, B. Kommandur, and M. Patyra. Designing a 3 GHz, 130 nm, Intel® Pentium®4 processor. *Symposium on VLSI Circuits, Digest of Technical Papers*, pages 130–133, 2002.

[6] Department of Software Engineering, University of Szeged. GCC code-size benchmark environment (csibe). `http://www.csibe.org/`.

| Benchmark | Bottleneck |
|---|---|
| bodytrack | Instruction variation between processors for various functions leads to different cycle counts for different functions (Figure 7). |
| canneal | A workload imbalance and large serial portion results in all threads but one spending 90% of their execution time in function swap_cost, while one thread only spends 35% of its total time. All spend the same number of cycles executing this function, but the fourth thread also executes additional functions. The swap_cost function generates a large amount of off-chip traffic, L2 cache misses and L2 tag snoops, resulting in CMPs being communication bound at high thread counts. |
| dedup | Thread contention and cache miss rates lead to a disparity in execution times. The keys_equal_fn function generates an unequal number of L2 misses between threads. Additionally, a lock is required in function hashtable_search(hashtable.c:207). As the program scales, more time is spent executing this function, and spinning on locks. |
| facesim | Increasing threads leads to increased bus contention. Main function Add_Force_Differential does not scale, due to 25% more outstanding read requests waiting on the bus. Array and object values used for calculations at VECTOR_3D.h:71 and MATRIX_3X3.h:136 require significant memory operations. |
| fluidanimate | Each thread spends 10% of their total cycles on lock/unlock synchronization code. Functions such as RebuildGridMT do not scale well due to bus contention, which slows down each thread by 10% compared to their single threaded counterpart. |
| streamcluster | 95% of compute cycles are spent finding the Euclidean distance between two points (streamcluster.cpp:157). Scaling is sensitive to memory speeds and bus contention when sourcing the operands for this operation. Although not graphed, it suffers from a 20+% DRAM page conflict rate. |

**Table 3.** PARSEC Benchmark Bottlenecks


[7] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[8] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile PCs in 45nm hi-$\kappa$ metal gate CMOS. *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 256–257, Feb. 2008.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE 4th Workshop on Workload Characterization*, pages 3–14, Dec. 2001.

[10] R. R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, and D. Newell. Exploring small-scale and large-scale cmp architectures for commercial java servers. In *Proc. IEEE International Symposium on Workload Characterization*, pages 191–200, Oct. 2006.

[11] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. IEEE/ACM 30th International Symposium on Microarchitecture*, pages 330–335, Nov. 1997.

[12] P. E. McKenney. *Differential Profiling*, volume 29, pages 219–234. John Wiley & Sons, Aug. 1999.

[13] F. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.

[14] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *Proc. 7th IEEE Symposium on High Performance Computer Architecture*, pages 85–96, Jan. 2001.

[15] U. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core 64-thread 64b power-efficient SPARC SoC. *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 108–109, Feb. 2007.

[16] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 63–73, Sept. 2004.

[17] H. Shikano, J. Shirako, Y. Wada, K. Kimura, and H. Kasahara. Power-aware compiler controllable chip multiprocessor. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 427–427, Sept. 2007.

[18] V. Weaver and S. McKee. Are cycle accurate simulations a waste of time? In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pages 40–53, June 2008.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd IEEE/ACM International Symposium on Computer Architecture*, pages 24–36, June 1995.